Pattern Matching

Outline and Reading

- Pattern matching algorithms
  - Brute-force algorithm
  - Boyer-Moore algorithm
  - Knuth-Morris-Pratt algorithm

Strings

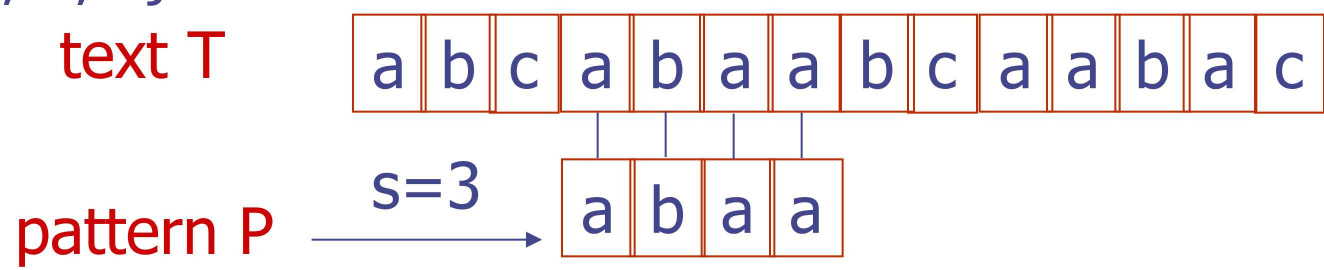
- A string is a sequence of characters
- Examples of strings:
  - Java program
  - HTML document
  - DNA sequence
  - Digitized image
- An alphabet Σ is the set of possible characters for a family of strings
- Example of alphabets:
  - ASCII
  - Unicode
  - {0, 1}
  - {A, C, G, T}
- Let P be a string of size m
  - A substring $P[i..j]$ of P is the subsequence of P consisting of the characters with ranks between i and j
  - A prefix of P is a substring of the type $P[0..i]$
  - A suffix of P is a substring of the type $P[i..m-1]$
- Given strings T (text) and P (pattern), the pattern matching problem consists of finding a substring of T equal to P
- Applications:
  - Text editors
  - Search engines
  - Biological research

String Matching

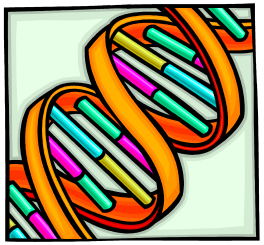
Given: Two strings T[1..n]] of length *n* and P[1..m] of length *m* over alphabet Σ
(The elements of *P* and *T* are characters drawn from a finite alphabet set *Σ*.)

Goal: Find all occurrences of P[1..m] "the pattern" in T[1..n] "the text".

Example: Σ = {a, b, c}



Brute-Force Algorithm



Input : Text S[0 . . . n-1] and pattern P[0 . . . m-1]
Output: Index of the left end of the first matching substring, or -1 if not found.

$i \leftarrow 1$
while $i \leq n - m + 1$ **do**
  $k \leftarrow 0$
  while $k \leq m - 1$ and $P[k] = S[i + k]$ **do**
    $k \leftarrow k + 1$
  if $k = m$ **then**
    return i
  $i \leftarrow i + 1$
return -1

Algorithm 1: $BruteForceMatcherLeftToRight(T, P)$: Finds P in S.

- The brute-force pattern matching algorithm compares the pattern ***P*** with the text ***T*** for each possible shift of ***P*** relative to ***T***, until either
  - a match is found, or
  - all placements of the pattern have been tried
- Brute-force pattern matching runs in time ***O***(***nm***)
- Example of worst case: ***T*** = ***aaa … ah***, ***P*** = ***aaah***
  - may occur in images and DNA sequences
  - unlikely in English text

Brute Force: Right to left

Input : Text S[0 . . . n-1] and pattern P[0 . . . m-1]
Output: Index of the left end of the first matching substring, or -1 if not found.

```
i ← m − 1
while i ≤ n − 1 do
    k ← 0
    while k ≤ m − 1 and P[m − 1 − k] = S[i − k] do
        k ← k + 1
    if k = m then
        return i − m + 1
    i ← i + 1
return −1
```

Algorithm 1: $BruteForceMatcherRightToLeft(T, P)$: Finds P in S.

Running time is $\Theta((n - m + 1)m)$.

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R
  B A R B E R
    B A R B E R
      B A R B E R
        B A R B E R
          B A R B E R
            B A R B E R
                 .
                 .
                 .
                               B A R B E R
```

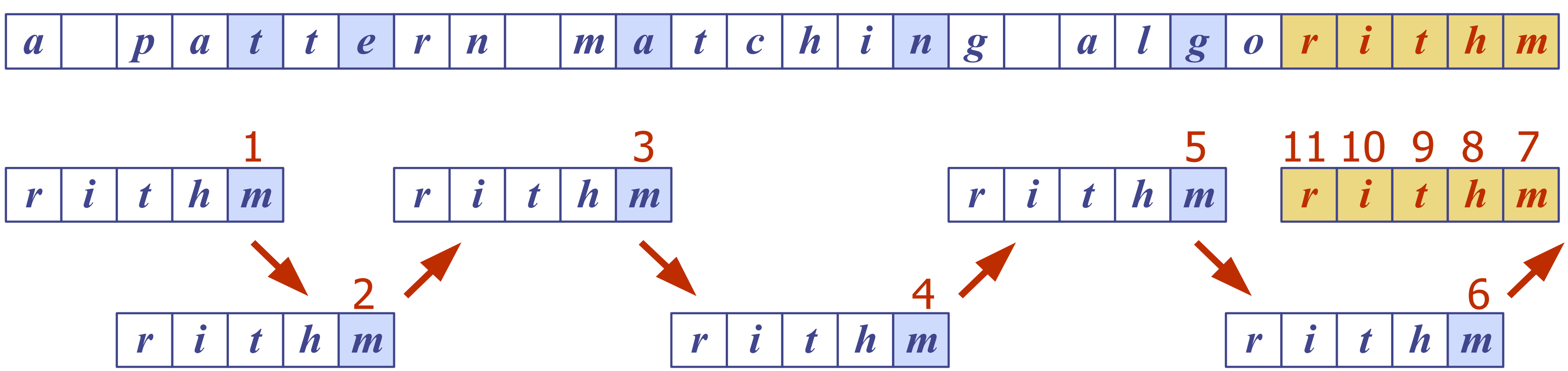
Boyer-Moore Heuristics

- The Boyer-Moore's pattern matching algorithm is based on two heuristics

  Looking-glass heuristic: Compare P with a subsequence of T moving backwards

  Character-jump heuristic: When a mismatch occurs at $T[i] = c$

  - If P contains c, shift P to align the last occurrence of c in P with $T[i]$
  - Else, shift P to align $P[0]$ with $T[i + 1]$
- Example



```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R         B A R B E R
          B A R B E R             B A R B E R
```

Last-Occurrence Function

- Boyer-Moore's algorithm preprocesses the pattern P and the alphabet Σ to build the last-occurrence function T mapping Σ to integers, where $T(c)$ is defined as
  - The distance between the last character and c's highest index in P.
  - m if no such index exists
- Example:
  - $\Sigma = \{a, b, c, d\}$
  - $P = abacab$

c	a	b	c	d
$T(c)$	1	4	2	6

- The last-occurrence function can be represented by an array indexed by the numeric codes of the characters
- The last-occurrence function can be computed in time $O(m + s)$, where m is the size of P and s is the size of Σ

The Boyer-Moore Algorithm

Input : Text S[0 . . . n-1] and pattern P[0 . . . m-1]
Output: Index of the left end of the first matching substring, or -1 if not found.

$T \leftarrow shiftTable(P, S)$
$i \leftarrow m - 1$
while $i \leq n - 1$ **do**
  $k \leftarrow 0$
  while $k \leq m - 1$ and $P[m - 1 - k] = S[i - k]$ **do**
    $k \leftarrow k + 1$
  if $k = m$ **then**
    return $i - m + 1$
  $skip \leftarrow max(1, T[S[i - k]] - k)$
  $i \leftarrow i + skip$
return -1

Algorithm 1: $BoyesMooreMatch(T, P)$: Finds P in S.

Case 1: 1 > T[a] - k

Case 2: 1 <= T[a] - k

Example

a	*b*	*a*	*c*	*a*	*a*	*b*	*a*	*d*	*c*	*a*	*b*	*a*	*c*	*a*	*b*	*a*	*a*	*b*	*b*

					1
a	*b*	*a*	*c*	*a*	*b*

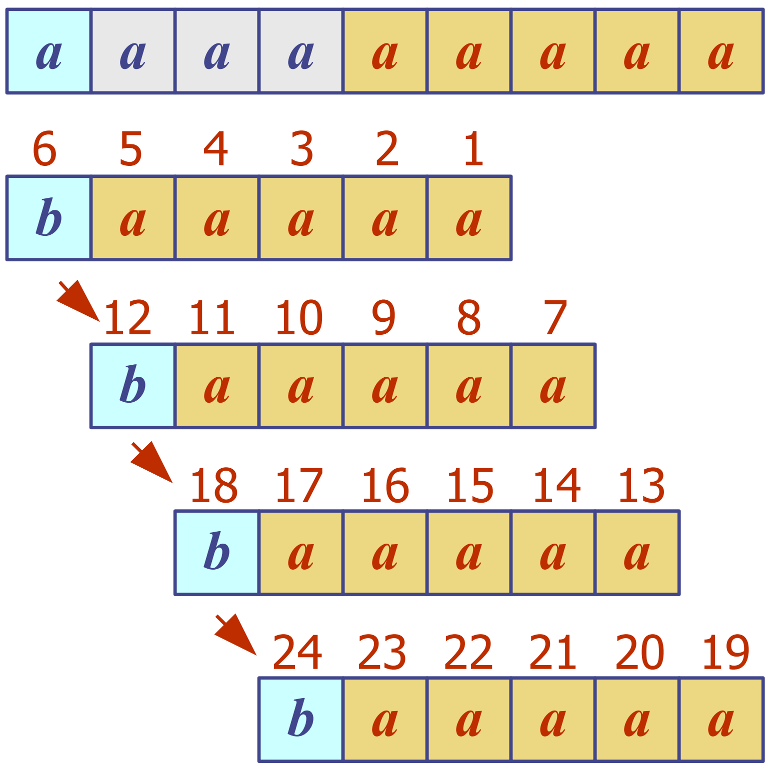
Example

Pattern = P = ababaca

String **a b a b a b a c a b a**

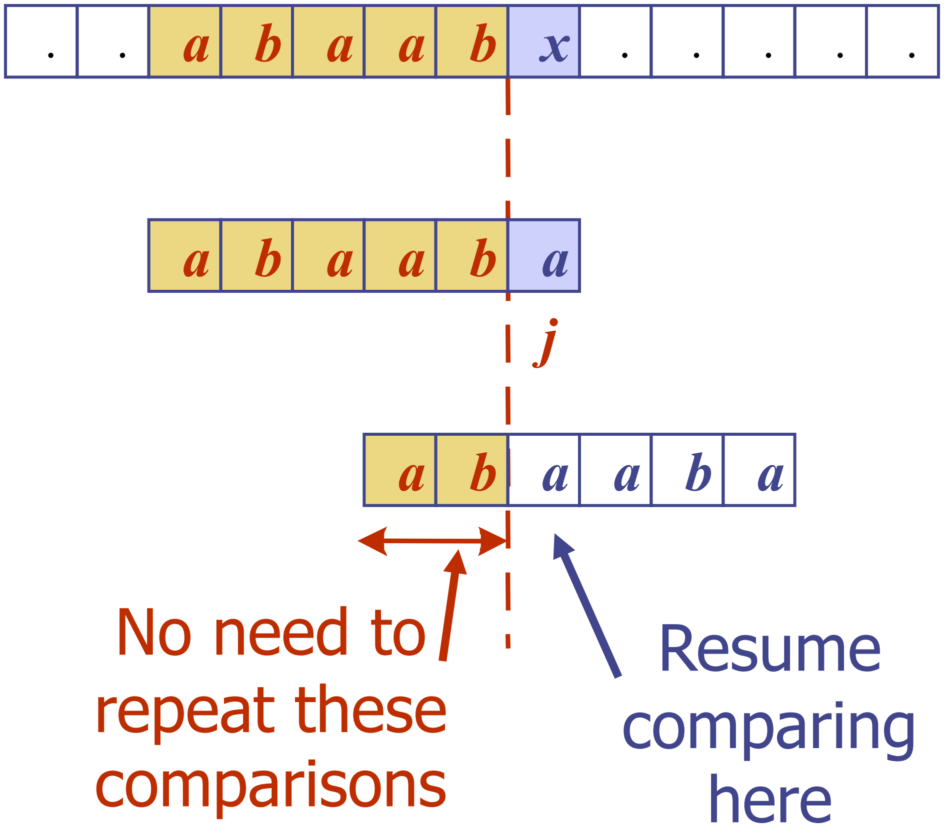
Analysis

- Boyer-Moore's algorithm runs in time $O(nm + s)$
- Example of worst case:
  - $T = aaa \ldots a$
  - $P = baaa$
- The worst case may occur in images and DNA sequences but is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text



The KMP Algorithm - Motivation

- Knuth-Morris-Pratt's algorithm compares the pattern to the text in **left-to-right**, but shifts the pattern more intelligently than the brute-force algorithm.
- When a mismatch occurs, what is the **most** we can shift the pattern so as to avoid redundant comparisons?
- Answer: the largest prefix of $P[0..j]$ that is a suffix of $P[1..j]$



Components of KMP algorithm

- The prefix function, Π

The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern 'p'. In other words, this enables avoiding backtracking on the string 'S'.

- The KMP Matcher

With string 'S', pattern 'p' and prefix function 'Π' as inputs, finds the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrence is found.

The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

Compute-Prefix-Function (p)

```
m ← length[p]                    //'p' pattern to be matched
Π[1] ← 0
k ← 0
    for q ← 2 to m
        do while k > 0 and p[k+1] != p[q]
            do k ← Π[k]
           If p[k+1] = p[q]
            then k ← k +1
           Π[q] ← k
    return Π
```

Example: compute Π for the pattern 'p' below:

p

a	b	a	b	a	c	a

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: q = 3, k = 0,
Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: q = 4, k = 1
Π[4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: q = 5, k =2

Π[5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: q = 6, k = 3

Π[6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	

Step 6: q = 7, k = 1

Π[7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	0	1

After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	0	1

The KMP Matcher

The KMP Matcher, with pattern ‘p’, string ‘S’ and prefix function ‘Π’ as input, finds a match of p in S.

Following pseudocode computes the matching component of KMP algorithm:

```
KMP-Matcher(S,p)
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                  //number of characters matched
5 for i ← 1 to n                         //scan S from left to right
6      do while  q > 0 and p[q+1] != S[i]
7              do  q ← Π[q]              //next character does not match
8          if p[q+1] = S[i]
9            then q ← q + 1              //next character matches
10         if q = m                      //is all of p matched?
11           then print "Pattern occurs with shift" i – m
12                q ← Π[ q]              // look for the next match
```

Note: KMP finds every occurrence of a ‘p’ in ‘S’. That is why KMP does not terminate in step 12, rather it searches remainder of ‘S’ for any more occurrences of ‘p’.

Illustration: given a String 'S' and pattern 'p' as follows:

S	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p	a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether 'p' occurs in 'S'.

For 'p' the prefix function, Π was computed previously and is as follows:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	0	1

Initially: n = size of S = 15;

m = size of p = 7

Step 1: i = 1, q = 0

comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p	a	b	a	b	a	c	a								

P[1] does not match with S[1]. ‘p’ will be shifted one position to the right.

Step 2: i = 2, q = 0

comparing p[1] with S[2]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p		a	b	a	b	a	c	a							

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

Comparing p[2] with S[3]   p[2] does not match with S[3]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Backtracking on p, comparing p[1] and S[3]

Step 4: i = 4, q = 0

comparing p[1] with S[4]   p[1] does not match with S[4]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Step 5: i = 5, q = 0

comparing p[1] with S[5]   p[1] matches with S[5]

S

b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p

a	b	a	b	a	c	a

Step 6: i = 6, q = 1

Comparing p[2] with S[6]   p[2] matches with S[6]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 7: i = 7, q = 2

Comparing p[3] with S[7]   p[3] matches with S[7]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 8: i = 8, q = 3

Comparing p[4] with S[8]   p[4] matches with S[8]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Step 10: i = 10, q = 5

Comparing p[6] with S[10] p[6] doesn't match with S[10]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p					a	b	a	b	a	c	a				

Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 12: i = 12, q = 5

Comparing p[6] with S[12]

p[6] matches with S[12]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Step 13: i = 13, q = 6

Comparing p[7] with S[13]

p[7] matches with S[13]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b
p							a	b	a	b	a	c	a		

Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Example

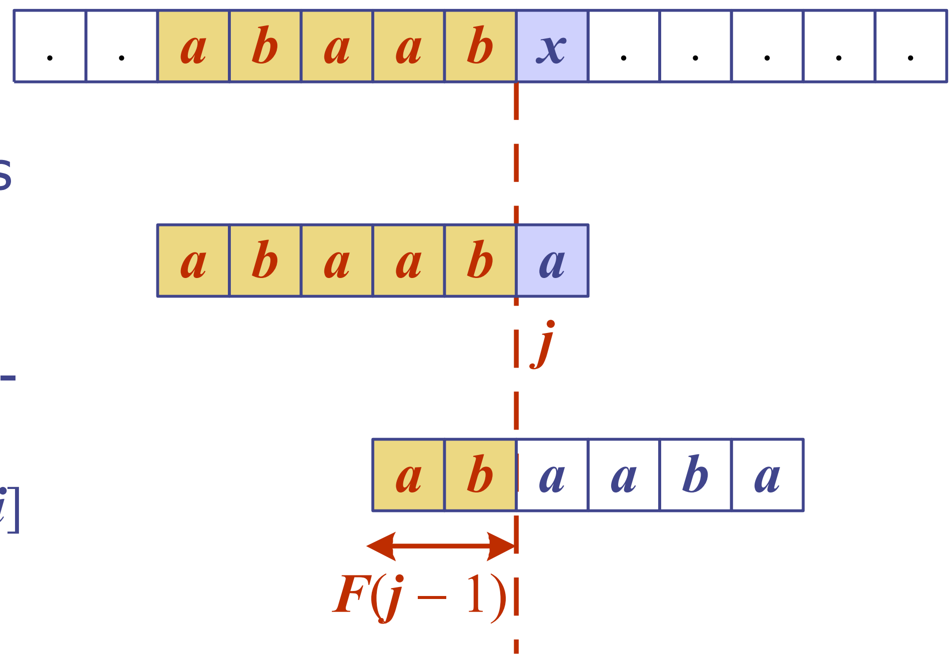
Prefix function of P= ATCACATCATCA ?

j	1	2	3	4	5	6	7	8	9	10	11	12
P:	A	T	C	A	C	A	T	C	A	T	C	A
f:	0	0	0	1	0	1	2	3	4	2	3	4

KMP Failure Function

- Knuth-Morris-Pratt's algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself
- The **failure function** $F(j)$ is defined as the size of the largest prefix of $P[0..j]$ that is also a suffix of $P[1..j]$
- Knuth-Morris-Pratt's algorithm modifies the brute-force algorithm so that if a mismatch occurs at $P[j] \neq T[i]$ we set $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

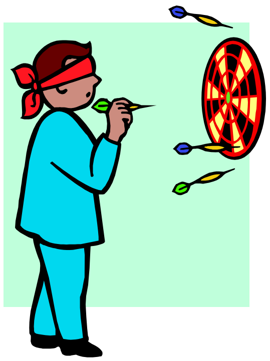


The KMP Algorithm

- The failure function can be represented by an array and can be computed in $O(m)$ time
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j - 1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop
- Thus, KMP's algorithm runs in optimal time $O(m + n)$

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```

Computing the Failure Function



- The failure function can be represented by an array and can be computed in $O(m)$ time
- The construction is similar to the KMP algorithm itself
- At each iteration of the while-loop, either
  - i increases by one, or
  - the shift amount $i - j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2m$ iterations of the while-loop

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 chars}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 { no match }
            i ← i + 1
```

Example

a	*b*	*a*	*c*	*a*	*a*	*b*	*a*	*c*	*c*	*a*	*b*	*a*	*c*	*a*	*b*	*a*	*a*	*b*	*b*

1	2	3	4	5	6
a	*b*	*a*	*c*	*a*	*b*

Running - time analysis

- Compute-Prefix-Function (Π)

```
1  m ← length[p]        //'p' pattern to be matched
2  Π[1] ← 0
3  k ← 0
4        for q ← 2 to m
5              do while k > 0 and p[k+1] != p[q]
6                    do k ← Π[k]
7                    If p[k+1] = p[q]
8                       then k ← k +1
9                    Π[q] ← k
10       return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is $\Theta(m)$.

- KMP Matcher

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6     do while  q > 0 and p[q+1] != S[i]
7             do  q ← Π[q]
8        if p[q+1] = S[i]
9           then q ← q + 1
10       if q = m
11         then print "Pattern occurs with shift" i – m
12                 q ← Π[ q]
```

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S'. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is $\Theta(n)$.